Runtime System Support for Parallel Iterative PDE Computations *

Vineet Ahuja Nikos Chrisochoides Induprakas Kodukula
Keshav Pingali

Abstract

In this paper, we describe optimizations for parallel iterative solvers of partial differential equations. The heart of such a solver is a matrix vector multiply routine. This routine performs computation on both local and non-local data. In this paper, we present optimizations which reduce the overall execution time of the computations on non-local data. In particular, we discuss optimizations that reduce the communication time at the expense of additional computation, and argue that this trade-off is beneficial. These optimizations have been integrated into a runtime system which is aimed specifically at scientific applications. This system is based on a substrate (DMCS) which is a Cornell implementation of the PORTS interface.

1 Introduction

Parallel iterative solvers for partial differential equations are very popular because they are fast and relatively easy to implement. The heart of such a solver consists of a matrix vector multiply (MVM) routine. The matrix is typically sparse, and is distributed among the processors participating in the computation. The global MVM is computed in parallel by having each processor do a local MVM in which some of the data must be communicated from other processors. In this paper, we discuss optimizations that reduce the time spent in communicating and computing with non-local data. There are two main optimizations — one that reduces the time processors spend *sending* data to other processors (at the expense of some additional computation), and another which reduces time the processors spend in receiving and computing with non-local data. The rest of the paper is organized as follows. In Section 2, we describe the runtime substrate that underlies our implementation. In Section 3, we describe the optimizations for reducing the time spent in sending the data. In Section 4, we describe the optimizations to reduce the time spent in receiving the data. Section 5 presents performance numbers. We conclude in Section **??**.

2 Runtime System

In this section, we describe the underlying runtime system substrate of our system, DMCS [4]. DMCS implements several APIs formulated by the PORTS (*POrtable Runtime Systems*) consortium. The first API, *ports_threads*, has already been agreed upon by the PORTS consortium. It comprises of a set of functions for lightweight thread management, modeled after a subset of the POSIX thread interface. An implementation of the

*This work supported by the Cornell Theory Center which receives major funding from the National Science Foundation, IBM corporation, New York State and members of the its Corporate Research Institute.

ports_threads interface is available from Argonne National Laboratory [5]. In addition, a set of functions have been specified for timing and event logging, using the high resolution, synchronized clocks available on many shared and distributed memory supercomputers. The timer package is thread safe, but not thread aware. In other words, a correct implementation of this specification can be used in a preemptive thread environment; however, the specification does not require threads. An extremely fast implementation of *ports_timing* is available from University of Oregon [10]. There is also a proposed API for communication, and for the integration of communication with threads[6]. Our DMCS implementation accomplishes the following.

- We provide a simple mechanism for reducing the scheduling latency of urgent remote service requests as well as the communication overhead associated with remote service requests for sparse, adaptive and irregular numerical computations.
- We isolate the interaction between threads and communication into a simple module called *control*, which is easy to understand and modify.
- We provide a global address space and a threaded model of execution that provides a common programming model for SMPs, clusters of SMPs and MPPs.
- We have a very lean and modular layer that allows us to "plug-and-play" with different module implementations from PORTS community.

2.1 Runtime System architecture

DMCS consists of three modules: (i) a *threads* module (ii) a *communication* module and (iii) a *control* module. The threads and communication modules are independent, with some clearly defined interface requirements, while the control module is built on top of the point-to-point communication and thread primitives. The *threads* module supports the primitives defined by the PORTS consortium, *ports_threads*. We are using the implementation provided by the Argonne group, PORTS0 [5], augmented by an extra routine: *ports_thread_create_atonce*. The efficient implementation of this routine is necessary to minimize the scheduling latency of certain urgent, *remote service requests* [2]. Clearly, this extension can be implemented on top of the existing *ports_threads* primitives provided by PORTS0(for example, using the thread priority attributes), but for efficiency reasons, we choose to implement it, whenever possible, directly on the underlying thread package. A prototype is implemented on top of the QuickThreads [7], which has been ported to a wide variety of workstation and PC architectures.

The communication module provides the necessary support for the implementation of a global address space over both shared and distributed memory machines. Collective communication primitives are not considered in this paper. In the future, we plan to evaluate and use the "rope" primitives introduced in [3]. Our communication abstraction is the *global pointer*. A global pointer essentially consists of a context number and a pointer to the local address space. The functionality of the communication module for point-to-point data-movement includes routines like *get/put* to initiate the transfer of data from/to a remote context to/from a local context. The interaction of the communication module and threads takes place in the *control* module which integrates the thread scheduler with the point-to-point communication mechanism. Figure 1 depicts the three modules of the DMCS and their interaction.

The *control* module provides support for remote procedure invocation, also known as *remote service requests* (RSRs). Remote procedures are either system/compiler procedures or user defined handlers. These handlers can be threaded or non-threaded. The threaded

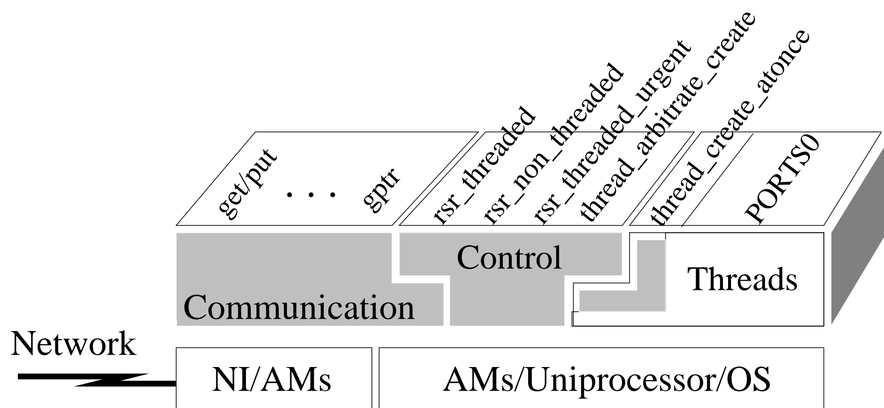


FIG. 1. *The architecture of the DMCS runtime system*

handlers can be either *urgent* (scheduled after a fixed quanta of time[1]) or can be *lazy* (scheduled only after all other computation and manager threads have been suspended or completed). The non-threaded handlers are executed either as the message is being retrieved from the network interface[2], or after the message retrieval has been completed [1]. Finally, the control module provides some limited support for simple load balancing by allowing the association of a window within which load on any processor can be balanced. This load-balancing support was chosen after experiments with the SplitThreads system [9].

3 Augmentation Optimization

We assume that parallel matrix-vector product Ax is implemented by distributing rows of the matrix A to processors. If row i is mapped to a processor p, we assume that element i of the vector x is mapped to that processor as well. To multiply row i by the vector, the processor needs the values of $x(j)$ for all j such that $A(i,j)$ is non-zero. This requires interprocessor communication. Before each processor sends out data owned by it to other processors, it must perform a gather operation. The gather operation aggregates all the data being sent to a processor into a contiguous buffer. If the data being sent out to different processors was completely disjoint, this gather would not be necessary. Every processor could reorder its own data such that all the data being sent out to a single processor was contiguous. The data could be sent directly without a gather operation. We propose an augmentation algorithm to do precisely this. We first illustrate this idea with a simple example. Next, we give a linear algebraic formulation of the procedure. Finally, we descibe how augmentation affects the rest of the computation.

3.1 Example

We first demonstrate the idea of augmentation using an example. Consider the local matrix vector multiply operation shown in Figure 2. The matrix is denoted by A and the vector by x. For the sake of simplicity that we are dealing with a dense product. Let x have 4

[1]The time interval of a timeslice or the time it takes for the next context-swithch of the current thread in the case of a non-preemptive scheduling environment.

[2]This works by overlapping computation and communication in the instruction level by interleaving the computation and flow of control that corresponds to the incoming message and the load/store operations needed to retrieve the message from the network interface.

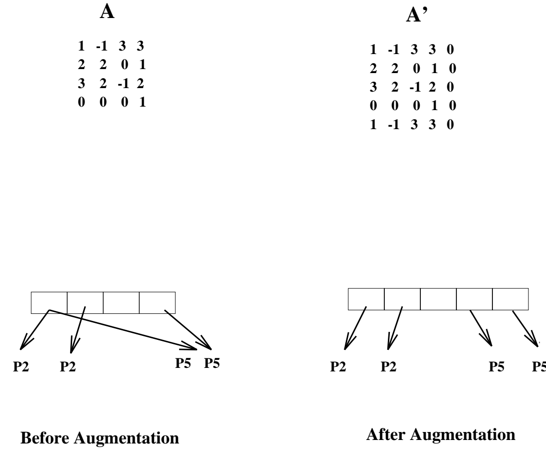


FIG. 2. *The augmentation optimization to eliminate gather operation on the sender side*

elements and A be 4-by-4. Assume that $x[1]$, $x[2]$ and $x[4]$ are communicated as shown in Figure 2. In this example, a gather is necessary since 1 is communicated to more than one processor. We eliminate the gather by performing an augmentation operation, as follows. Since $x[1]$ is communicated twice, we create a copy for it by making one more element in x. We want to maintain the invariant that after the local matrix vector multiply is complete $x[1]$ and $x[5]$ have the same value. This is done by augmenting the matrix A as well to ensure the correct computation of x[5] in the matrix-vector product phase. This is done by augmenting the matrix A by 1 row and 1 column. The row added is a copy of the row 1, which ensures that the computed value of $x[5]$ would be the same as that of $x[1]$. The column added is however filled with zeros. This is because we do not want to use $x[5]$ in any computation. (we do not need to, anywhere it could be used, we can use $x[1]$). Thus a 4-by-4 matrix is converted to a 5-by-5 matrix. After the augmentation, we note that $x[1]$ and $x[2]$ are now being sent to P2 and $x[4]$ and $x[5]$ are being sent to P5. In fact, the data being sent to the processors is also contiguous and distinct. However, in general, augmentation will only ensure that any element will be sent to at most one processor. A permutation of the vector and the matrix is still necessary to ensure that all the data being sent to any one processor is contiguous. We refer to elements such as $x[5]$ as **duplicated elements**.

3.2 Linear algebra framework

In this section, we model the augmentation procedure described in the previous section as a linear transformation on the data. The motivation for doing this is to explore the possibility that a compiler can do this transformation automatically.

We derive the linear transformation that describes augmentation. The linear transformation is composed of two parts - transformation on the code and transformation on the data. We examine first the transformation on the data.

3.2.1 Data transformation The data transformations that interest us are those that apply to x and to A. Assume that x is a vector with n elements and A is an $n \times n$ matrix. Let i_a denote the index of the element of x that is being augmented. Without loss of generality, assume that the duplicate of $x[i_a]$ is the last element of the new vector. This is

the complete transformation for x. The transformation for A is more complex. Let A_t be the matrix after the data transformation is applied to A. Then A_t can be written as PAQ, where P is an n+1-by-n matrix and Q is an n-by-n+1 matrix. The first n rows of P are the same as that of the identity matrix and the last row has a single nonzero at column i_a. The first n columns of Q are those of the identity matrix and the last column has all zeros.

In the example in the previous section, the matrix A_t can be written as

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} * A * \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Finally, we note that the augmented matrix is singular. This is obvious because the rank of the final matrix is the same as the rank of the initial transformation. In general, this could be a problem, because several algorithms which employ the matrix vector multiply routine require the matrix to be nonsingular. However, there is a fairly simple solution to this. Instead of just using a linear transformation as seen above, we use an affine transformation. In other words, our transformed matrix can be written as $PAQ + H$, where P and Q are as above and H is of the following form. H is an n+1-by-n+1 matrix and only its last row is nonzero. The last row has 2 nonzero elements: -1 in column i_a and 1 in column $n+1$. The transformed matrix for our running example can be written as follows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} * A * \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

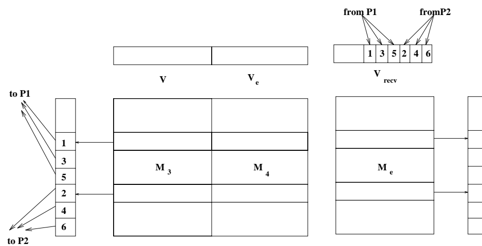
3.3 Why Augmentation is a good idea

We now give some back of the envelope calculations for when augmentation can be a good idea. Although we have explained the procedure in the context of dense matrices, we expect the benefit to be more significant in the case of sparse matrices. The reason is that if A is sparse, then the recomputation of elements of the augmented vector are less expensive. For example, consider a matrix that represents the adjacencies of a two-dimensional mesh. Let n represent the number of points of the mesh in each dimension. The number of the points in the mesh is the n^2. The matrix that represents the adjacency of the mesh is extremely sparse - it has at most 5 non zero elements per row or column. This means that recomputation of an element of x takes at most 10 floating point operations (5 multiply and 5 add operations). Now suppose that this mesh is distributed onto a square grid of processors with p processors in each dimension. Let m denote the number of points in each dimension of the portion of the mesh that is allocated to each processor. ($m = n/p$). Each processor communicates with upto 4 neighbours (to the left and right, as well as above and below). The volume of data communicated is equal to the number of points on the boundary, which is equal to $4 * m$. This is also the amount of work done to gather the data into the communication buffers. However, only the 4 corner elements need to be sent to more than one processor. They each have to be sent to 2 processors. Consequently, the augmentation procedure will augment each of the corner elemennts once. Thus in every execution of the matrix vector multiply routine, 4 * 10 = 40 additional floating point

operations need to be performed. However $4 * m$ amount of copying will be saved in the send routine. For large mesh sizes, this can be a substantial saving.

3.4 Avoiding redundant computations

An issue that comes up after augmentation is the avoidance of redundant computations. In particular, we must ensure that the results computed by the original program are not affected by the augmentation. For example, if the original program was computing the 2-norm of the vector x after the vector product, then we should do the computation to produce the same value. In this section, we show what additional data structures are needed by the augmentation procedure to be able to handle such computations. We use the 2-norm example. Let us assume that after the completion of the matrix vector product, we calculate the 2-norm of x using the following code.



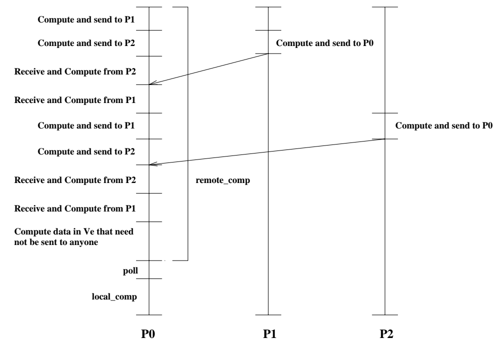


FIG. 3. *Receive handlers*

```
sum = 0;
for (i = 0; i < length(x) ; i++) {
  sum += x[i] * x[i];
}
```

Since x now has more elements than before, (some of them duplicated), it is clear that we will compute the wrong 2-norm if we follow this loop naively. There are 2 ways to tackle this problem. The first is to execute the loop fully for the new vector and then to subtract out the contribution from the duplicated elements. However, this means doing the redundant computation twice. A better solution is to do the following. We reorder the elements of x as follows. We store all the local elements of x contiguously. This includes all elements of x that are owned locally (some of these elements are sent out to other processors). In this storage, we first store all elements of x owned locally that are not sent to any of the other processors. We call these the **interior** points. Next we store the data being sent to various processors(which we call **local interface** points. The data being sent to each processor is stored contiguously. This data is partitioned into two contiguous blocks - the first block contains all the data being sent to this processor that was present in the initial vector x. The second block contains all the data being sent to that processor which is made up of **duplicated elements**. For every neighbouring processor, we now need to maintain an additional data structure that stores the boundary between non duplicated and duplicated elements for that processor. We also need to store a pointer to the end of the interior points. Given all this information, we can now rewrite the above 2-norm loop as follows:

```
sum = 0
for (i = 0; i < interior (x); i++) {
  sum += x[i] * x[i];
}
for (i = 0; i < nneighbors; i++) {
  for (j = neighbor(i).send.start;
       j < neighbor(i).duplicate.start;
       j++)
    sum += x[j] * x[j];
}
```

It can be shown by a simple argument that no duplicated element is involved in the computation of sum and that every non-duplicated element is involved in its computation precisely once. Thus, we can compute the 2-norm of x with no redundant computation. A similar strategy can be applied to the evaluation of dot products of x and some other vector.

4 Receive handlers

When the non-local portion of the right-hand side vector is received, it is first copied to the appropriate receive buffers. At a later time, the processor visits the receive buffer to perform computation on this data. In a system such as the SP-2 with no DMA, this means that non-local data passes through the cache twice (once when it is copies to receive buffers, and later when it is actually used in computation). To better utilize the network interface and cache, we restructure the computation in the manner desribed next: Each box in the ans vector is of the same size as the the payload in an Active Message packet (twenty five doubles). The computation now changes so that the processor computes just the first twenty five values that have to be sent to P1 and immediately sends them. It then computes the first twenty five values that go to P2 and sends those. Figure 3 explains this.

Sending data in these packet-sized sections as opposed to a send of a single big vector as in the earlier case, gives the application a smaller overhead. Also unlike the previous case, since the processor sends data immediately after it is computed, the data has to be present in the cache and we avoid having to retrieve data from main memory. The reason a processor sends data cyclically to other processors is to avoid flooding one processor with a number of messages, leaving other processors idle. The processor will send another processor data to compute giving it time to finish the computation and only then send the next packet. The loops are scheduled so that no processor is ever waiting for data. At the receive end, the processors receive packet sized buffers and immediately multiply them with the corresponding columns of M_e. Since the data was just received at the network interface it will be in the cache only once and we avoid the cache miss a conventional approach would have. Also, there is no longer need to store the received data anywhere since there is no more use for it. This saves another write to main memory. The data is simply discarded. The network is polled implicitly every time the processor does a packet send. After all the packets are sent, the processor will wait in a poll loop till all the expected packets have arrived.

5 Performance

Figure 5 shows the performance improvements due to the optimizations we have outlined. The overall savings on application time spent in managing (communication + computation)

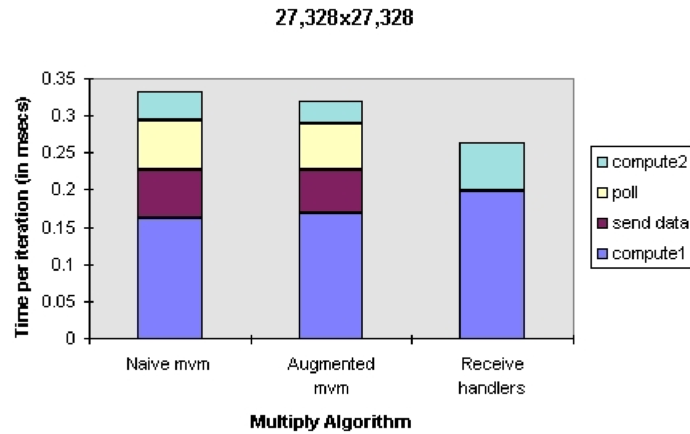


FIG. 4. *Performance improvement due to our optimizations*

non-local data is approximately 20%. This savings remains at about the same level as the problem size is increased.

References

[1] Thorsten von Eicken, Davin E. Culler, Seth Cooper Goldstein, and Klaus Erik Schauser, Active Messages: a mechanism for integrated communication and computation *Proceedings of the 19th International Symposium on Computer Architecture, ACM Press*, May 1992.

[2] N. Chrisochoides and Juan Miguel del Rosario, A Remote Service Protocol for Dynamic Load Balancing of Multithreaded Parallel Computations. Poster presentation in Frontiers'95.

[3] N. Sundaresan and L. Lee, An object-oriented thread model for parallel numerical applications. *Proceedings of the 2n Annual Object-Oriented Numerics Conference - OONSKI 94, Sunriver, Oregon, pp 291-308*, April 24-27 1994.

[4] N. Chrisochoides, I. Kodukula, K. Pingali, *Data Movement and Control Substrate for parallel scientific computing*, to appear in CANPC'97.

[5] The PORTS Consortium, *PORTS Level 0 Thread Modules from Argonne/CalTech*, ftp://ftp.mcs.anl.gov/pub/ports/

[6] The PORTS Consortium, *A Proposal for PORTS Level 1 Communication Routines*, http://www.cs.uoregon.edu/research/paracomp/ports

[7] D. Keppel, *Tools and Techniques for Building Fast Portable Threads Package*, UW-CSE-93-05-06, Technical Report, University of Washington at Seattle, (1993).

[8] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken and Katherine Yelick. Parallel Programming in Split-C. Supercomputing'93.

[9] Veena Avula. SplitThreads - Split-C threads. Masters thesis, Cornell University. 1994.

[10] Portable Clock and Timer Module from Oregon, http://www.cs.uoregon.edu/research/paracomp/ports